Techniques et applications du traitement de la langue naturelle
Travail pratique 3

Classification de texte et analyse de sentiments dans les conversations

Équipe 14:
Diego Silva Pires
NI: 111220136

14 décembre 2018

Introduction

Ce travail consiste à faire l'analyse de conversations afin de détecter les émotions qui y sont présentés selon les données fournies dans le cadre de la compétition *EmoContext* de Microsoft.

Un fichier texte mis à la disposition a été utilisé dans ce projet pour entraîner les classificateurs. Le fichier contient un échange de messages ainsi que l'émotion qui représente le mieux la conversation.

```
id  turn1   turn2   turn3   label
0   Don't worry  I'm girl   hmm how do I know if you are    What's ur name? others
1   When did I? saw many times i think -_-  No. I never saw you angry
2   By  by Google Chrome    Where you live  others
3   U r ridiculous  I might be ridiculous but I am telling the truth.   U little disgusting
whore   angry
4   Just for time pass  wt do u do 4 a living then  Maybe   others
5   I'm a dog person    youre so rude   Whaaaat why others
```

Un deuxième fichier a été fourni. Par contre, dans celui-ci, il n'y a pas d'émotion. Il faut alors prédire l'émotion correspondante selon le meilleur classificateur.

Le code résoudre cette problématique est sur Github:

https://github.com/DiegoPires/IFT-7022/tree/master/TP3

Python 3.6.2 a été utilisé avec la virtualisation de répertoire (*virtualenv*). Toutes les dépendances du projet se retrouvent dans le fichier *requirements.txt*.

Des instructions pour initialiser le répertoire virtuel et le code se retrouvent sur le fichier *README.md* dans GitHub.

1. Solution

Deux solutions ont été réalisées pour prédire les émotions des conversations. La première solution utilise des classificateurs de la bibliothèque SkLearn alors que la deuxième utilise la bibliothèque Keras avec Tensor Flow.

Pour appliquer ces solutions, plusieurs tests ont été faits avec des configurations différentes pour les deux librairies en utilisant le fichier d'entraînement.

Ces tests ont ensuite permis de trouver le classificateur pour chacune avec la meilleure précision afin de l'utiliser pour prédire les conversations du fichier sans émotion.

Dans les deux fichiers, la conversation est divisée en trois tours, mais pour les deux solutions proposées, le tout a été unifié dans une seule phrase avec des séparateurs.

Lors que le programme est exécuté, des fichiers avec les résultats sont créés dans le répertoire "*results"* dans GitHub. Pour chaque exécution du programme, ces fichiers sont supprimés et recréés.

Le temps total d'exécution pour tous les tests était de 3 heures, dont 50 minutes pour les classificateurs SkLearn et le restant pour les classificateurs Keras dans un ordinateur Mac avec une GPU intégrée.

1.1. SkLearn

La bibliothèque SkLearn a été choisie puisqu'elle est facile à utiliser et reconnue pour la réalisation de ce type de classification.

En ce qui concerne les classificateurs de la bibliothèque, les classificateurs suivants ont étés choisis pour les tests: *MultinomialNB*, *LogisticRegression*, *SGDClassifier*, *LinearSVC*, *SVC.*

Ces classificateurs sont efficients pour ce type de problématique. Ils ont entre autres été efficaces lors de la réalisation d'autres projets avec une classification semblable.

Ensuite, les attributs utilisés pour tester sont: avec ou sans mot outils, fréquence minimale, fréquence maximale, compte de mots, présence de mot, longueur n-gramme, compte de emojis et compte de sentiment positif/négatif.

L'attribut compte de emojis ajoute *une feature* avec le compte de quelques emojis très présents dans les phrases.

L'attribut de sentiment positif/négatif analyse chaque mot d'une phrase et ajoute dans un compte de chaque sentiment dans une nouvelle *feature*.

Pour la réalisation des tests, deux classes ont été créées pour faciliter la réutilisation du code:

```
class ClassifierTestSet:
   def __init__(self, name, classifier, stop_words=None, max_df=1.0, min_df=1,
use_Tfid=False, binary=False, ngram_range=(1, 1), apply_count_features=False,
apply_sentiment_features=False):
     # Code enlevé par souci de concision

class SkLearnClassifier(Classifier):
   def __init__(self, data_train, data_test, target_train, target_test, target_names):
      # Code enlevé par souci de concision
   def train_classifier(self, classifierTest, verbose):
      # Code enlevé par souci de concision
   def predict(self, text):
      # Code enlevé par souci de concision
```

La classe *ClassifierTestSet* fonctionne uniquement comme un objet de transfert de données (*Data transfer object* en anglais) qui contient un objet représentant du classificateur *SkLearn* à tester et quelques paramètres possibles lors de sa création.

C'est dans la classe *SkLearnClassifier* que le travail de classification est fait. Lors de la création d'un objet de ce type, on passe les donnés d'entraînement.

Ensuite, en appelant la méthode *train_classifier* et passant un objet *ClassifierTestSet* avec les paramètres désirés, notre modèle est créé et testé avec les donnés d'entraînement . De plus, il possède un valeur de précision.

Enfin, la méthode *predict* sert à réaliser des prédictions d'un texte donné selon le classificateur créé précédemment.

De cette façon, il est possible de créer facilement une suite de tests avec nos classificateurs choisis et une variété de paramètres différents.

```
classifiers = [
	ClassifierTestSet('MultinomialNB', MultinomialNB(), stop_words=None, max_df=1.0,
min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('LogisticRegression', LogisticRegression(), stop_words=None,
max_df=1.0, min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('SGD ', SGDClassifier(max_iter=1000), stop_words=None, max_df=1.0,
min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('LinearSVC ', LinearSVC(random_state=0, tol=1e-5), stop_words=None,
max_df=1.0, min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('SVC', SVC(kernel="linear", C=0.025), stop_words=None, max_df=1.0,
min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('SVC', SVC(kernel="rbf", C=0.025), stop_words=None, max_df=1.0,
min_df=1, use_Tfid=False, binary=False),
	ClassifierTestSet('LinearSVC ', LinearSVC(random_state=0, tol=1e-5), stop_words=None,
max_df=1.0, min_df=1, use_Tfid=True, binary=False, ngram_range=(1,2),
apply_count_features=True, apply_sentiment_features=True),
	# Code enlevé par souci de concision
]
```

Finalement, en réalisant les tests dans une boucle, c'est possible de créer tous les tests désirés et évaluer le meilleur avec les résultats obtenus.

```
for classifier in classifiers:
	skLearnClassifier = SkLearnClassifier(data_train, data_test, target_train,
target_test, target_names)
	skLearnClassifier.train_classifier(classifier, verbose)
```

Comme la réalisation des tests est plutôt lente, l'ajout des attributs de compte de emojis et compte de mots avec sentiments positifs/négatifs ont été ajoutés à la fin seulement pour six classificateurs qui apparaissent dans le top 10 des tests sans ces attributs ajoutés.

À la fin des tests, un fichier est créé avec tous les résultats. Ce fichier se retrouve dans *results/sklearn_classifiers.txt.*

Grâce à cette flexibilité de créer des cas de tests avec des paramètres différents, il a été possible de créer 138 tests uniques lors de l'évaluation du meilleur classificateur.

Le tableau suivant présente le top 10 des classificateurs selon leur précision:

Classifier	Mot outils	Freq. min	Freq. max	tfi-df	binaire	ngram	Emoji compte	Sentiment	Précision
LinearSVC	None	1	1	TRUE	TRUE	(1, 2)	TRUE	TRUE	0.842838196
SGD	None	1	1	TRUE	TRUE	(1, 2)	TRUE	TRUE	0.837201592
LinearSVC	None	1	1	TRUE	FALSE	(1, 2)	TRUE	TRUE	0.835377984
LogisticRegression	None	1	1	FALSE	FALSE	(1, 2)	TRUE	TRUE	0.834051724
SGD	None	1	1	FALSE	FALSE	(1, 2)	TRUE	TRUE	0.829575597
LogisticRegression	None	1	1	FALSE	FALSE	(1, 1)	TRUE	TRUE	0.8204575 6
LinearSVC	None	1	1	TRUE	TRUE	(1, 2)	FALSE	FALSE	0.807029178
LinearSVC	None	1	1	TRUE	FALSE	(1, 2)	FALSE	FALSE	0.805537135
LogisticRegression	None	1	1	FALSE	FALSE	(1, 2)	FALSE	FALSE	0.8012267 9
SGD	None	1	1	FALSE	FALSE	(1, 2)	FALSE	FALSE	0.797911141

Avec les résultats obtenus, le classificateur avec la meilleure précision est sélectionné pour faire la prédiction des conversations sans émotion. Les prédictions des conversations sans émotion se retrouvent dans un fichier disponible ici: *results/sklearn_prediction.txt.*

Le tableau ci-dessous présente les 10 premières prédictions:

Prediction	Sentence
angry	Then dont ask me YOURE A GUY NOT AS IF YOU WOULD UNDERSTAND IM NOT A GUY FUCK OFF
others	Mixed things such as?? the things you do. Have you seen minions??
happy	Today I'm very happy and I'm happy for you ❤ I will be marry
others	Woah bring me some left it there oops Brb

others	it is thooooo I said soon master. he is pressuring me
others	Wont u ask my age?? hey at least I age well! Can u tell me how can we get closer??
angry	I said yes What if I told you I'm not? Go to hell
others	Where I ll check why tomorrow? No I want now
others	Shall we meet you say- you're leaving soon...anywhere you wanna go before you head? ?
sad	Let's change the subject I just did it .I. You're broken
others	Your pic pz thank you X-D wc

1.1. Keras

La bibliothèque Keras a été choisie pour réaliser une comparaison entre une bibliothèque de classification classic (*SkLearn*) et une bibliothèque avec *deep learning*.

Ce choix a été fait parce que Keras est très adopté dans le domaine et entre autre, pour résoudre ce type de problématique.

Pour les tests avec Keras, quelques classes et méthodes ont été créées pour rendre plus faciles et réutilisables ces tests. On peut notamment parler de la classe *KerasClassifier* et de cinq méthodes responsables de créer cinq modèles différents avec variations.

```
def get_simple_keras_classifier(name, data_dto, count_vectorizer_dto=None, verbose=False):
   # Code enlevé par souci de concision
def get_denser_keras_classifier(name, data_dto, count_vectorizer_dto=None, verbose=False):
   # Code enlevé par souci de concision
def get_denser_keras_classifier_with_tokenizer(name, data_dto, keras_tokenizer_dto=None
verbose=False):
   # Code enlevé par souci de concision
def get_keras_with_word2vec(name, data_dto, extra_param_dto=None, verbose=False):
   # Code enlevé par souci de concision
def get_keras_with_word2vec_denser(name, data_dto, extra_param_dto=None, verbose=False):
   # Code enlevé par souci de concision
```

Les cinq méthodes font la création d'un Keras avec un modèle séquentiel (*Sequential* model*).* Ce qui diffère entre elles c'est la quantité de *layers* utilisée et la façon de vectoriser le texte.

Les deux premières méthodes utilisent la classe *CountVectorizer* de la bibliothèque *SkLearn* pour la vectorisation*,* ce qui permet de tester avec les mêmes attributs utilisés dans les tests précédents des classificateurs *SkLearn*, c'est-à-dire avec ou sans mot outils, fréquence minimale, fréquence maximale, compte de mots, présence de mot et longueur n-gramme.

La différence entre ces deux méthodes est que la première utilise uniquement deux couches de traitement: une d'entrée et une de sortie.

La deuxième méthode utilise quant à elle cinq couches: trois d'activation et deux pour éviter le surapprentissage.

La troisième méthode fait la création d'un Keras avec la classe *Tokenizer* de la bibliothèque Keras pour la vectorisation. Elle utilise huit couches: six de traitement et deux pour éviter le surapprentissage.

Les deux dernières méthodes font la création du modèle avec la bibliothèque Word2Vec comme responsable de la vectorisation des phrases.

Dans ces deux cas, la différence est aussi le nombre de couches, la première méthode utilise seulement deux couches, une d'entrée et une de sortie ,alors que l'autre méthode utilise huit couches: six de traitement et deux pour éviter le surapprentissage.

Pour exécuter le tout, on utilise la classe *KerasClassifier* en passant le méthode voulu pour créer le classificateur, plus des paramètres optionnels si nécessaire:

```
classifier_test = [
       KerasClassifierTestSet(name='simple', creation_method=get_simple_keras_classifier,
data_dto=data_dto, extra_param=None, verbose=verbose),
       KerasClassifierTestSet(name='denser', creation_method=get_denser_keras_classifier,
data_dto=data_dto, extra_param=None, verbose=verbose),
       KerasClassifierTestSet(name='denser_and_tokenizer_binary',
creation_method=get_denser_keras_classifier_with_tokenizer, data_dto=data_dto,
extra_param=KerasTokenizerDTO(None, True, ' ', False, 'binary'), verbose=verbose),
       KerasClassifierTestSet(name='denser_and_tokenizer_count',
creation_method=get_denser_keras_classifier_with_tokenizer, data_dto=data_dto,
extra_param=KerasTokenizerDTO(None, True, ' ', False, 'count'), verbose=verbose),
       KerasClassifierTestSet(name='denser_and_tokenizer_tfidf',
creation_method=get_denser_keras_classifier_with_tokenizer, data_dto=data_dto,
extra_param=KerasTokenizerDTO(None, True, ' ', False, 'tfidf'), verbose=verbose),
       KerasClassifierTestSet(name='denser_and_tokenizer_freq',
creation_method=get_denser_keras_classifier_with_tokenizer, data_dto=data_dto,
extra_param=KerasTokenizerDTO(None, True, ' ', False, 'freq'), verbose=verbose),
       KerasClassifierTestSet(name='word2vec',
creation_method=get_keras_with_word2vec_denser, data_dto=data_dto, extra_param=None,
verbose=verbose),
       KerasClassifierTestSet(name='word2vec_denser',
creation_method=get_keras_with_word2vec_denser, data_dto=data_dto, extra_param=None,
verbose=verbose),
   ]

   for test in classifier_test:
       classifier = test.execute()
```

Parce que le code a été exécuté dans un ordinateur Mac sans une GPU dédié, les tests se sont avérés très lents pour Keras.

Une mesure palliative appliquée dans le code a été de réaliser l'enregistrement des modèles dans le répertoire "keras_models". Si le système est exécuté et qu'il n'y a pas de modèle enregistré dans ce répertoire, le même est créé. Si le modèle existe déjà, il est réutilisé.

Il y a présentement seulement les deux premiers modèles enregistrés dans GitHub parce que les fichiers créés sont trop volumineux, ce qui n'est pas recommandé pour GitHub et peut être un inconvénient pour le partage de ce projet en tant que code Open Source.

Le problème de performance a aussi limité la variété d'attributs fournis pour la création des modèles.

Alors, au lieu de tester une variété d'attributs différents, la priorité a été de réaliser des modèles avec des méthodologies différentes. Cela justifie les plusieurs méthodes de création différents.

Ensuite, la contrainte du temps d'exécution a limité le nombre de tests à seulement 8 différents, soit un pour chaque méthode, sauf la méthode *get_denser_keras_classifier_with_tokenizer,* où une différence est notable avec le mode de vectorization: *binary, count, tf-idf* et *frequency*.

Après la réalisation des tests, un fichier est créé avec tous les résultats. Ce fichier se retrouve dans *results/keras_classifiers.txt.*

Le tableau suivant présente la liste des classificateurs ordonnés selon leur précision:

Nom du classificateur	**Précision**
word2vec_denser	0.848972148541114
word2vec	0.84375
denser_and_tokenizer_binary	0.7944297082228117
denser_and_tokenizer_count	0.7931034482758621
denser_and_tokenizer_tfidf	0.7853116710875332
denser_and_tokenizer_freq	0.7798408488063661
denser	0.7660809018567639
simple	0.690815649867374

Le classificateur avec la meilleure précision est choisi. Il est ensuite utilisé pour la prédiction des conversations sans émotion. Un fichier avec les résultats est disponible. Il se retrouve ic: *results/keras_prediction.txt.*

Le tableau ci-dessous présente les 10 premières prédictions:

Prediction	Sentence
angry	Then dont ask me YOURE A GUY NOT AS IF YOU WOULD UNDERSTAND IM NOT A GUY FUCK OFF
others	Mixed things such as?? the things you do. Have you seen minions??
happy	Today I'm very happy and I'm happy for you ❤ I will be marry
sad	Woah bring me some left it there oops Brb
sad	it is thooooo I said soon master. he is pressuring me
others	Wont u ask my age?? hey at least I age well! Can u tell me how can we get closer??
angry	I said yes What if I told you I'm not? Go to hell
others	Where I ll check why tomorrow? No I want now
others	Shall we meet you say- you're leaving soon...anywhere you wanna go before you head? ?
others	Let's change the subject I just did it .I. You're broken
others	Your pic pz thank you X-D wc

2. Conclusion

Les deux bibliothèques utilisées pour résoudre la problématique ont obtenu une précision au-dessus de 80%, au moins avec un des tests créés.

Comparativement aux résultats trouvés dans le site de la compétition, qui sont affichés dans le tableau ci-dessous, si la précision de 0.84 trouvée lors des tests se maintient pour les prédictions, le résultat se démarquera de ceux répertoriés:

Username	Score
jogonba2	0.7723
mfzszgs	0.7679
rafalposwiata	0.7660

Une dernière mesure calculée dans le code, est la différence de prédiction entre le meilleur classificateur des deux méthodes. Il est intéressant de remarquer que même si la meilleure précision des deux est de 0.84, la différence entre les deux est de 0.7662.

En ce qui concerne les limites du projet, Keras est une bibliothèque facile à s'initier à priori avec l'entraînement pour le *deep learning*. Par contre, elle demande plus de lectures pour approfondir et mieux comprendre toutes ses possibilités. Aussi, il est préférable de l'utiliser avec un ordinateur puissant et un GPU dédié pour faciliter son apprentissage étant donné les longs délais possibles pour l'application.

Néanmoins, pour ce type de problème, on n'a pas vraiment besoin d'utiliser *deep learning* puisqu'on a été capable d'obtenir un bon résultat aussi avec *SkLearn*.

Ce qui est positif avec *Keras* est qu'il est possible de sauvegarder le modelé créé. Il peut alors être utilisé dans d'autres langages, ou même avec un nouveau corpus facilement.

En faisant une analyse pour approfondir des classificateurs SkLearn, on note que les meilleurs classificateurs sont *LinearSVC, SGD* et *LogisticRegression*.

L'ajout des attributs de compte de emoji et compte de mots sont les critères qui donnent les meilleures résultats, et ce peu importe le classificateur.

La suppression des *stop words* et l'utilisation des fréquences minimales et maximales de mots sont des critères avec des pires résultats de précision.

Le restant des attributs ont varié en efficience selon le classificateur et l'utilisation conjointe d'autres paramètres.

Du côté de Keras, sauf pour le modèle qui utilise Word2Vec, l'utilisation de plusieurs couches semble être l'idéal pour l'entraînement.

Finalement, c'est l'utilisation de Word2Vec qui a amené les meilleures résultats. Cependant, on ne peut pas conclure qu'il est toujours le meilleur, parce qu'il n'y a pas eu beaucoup d'exploration lors de l'utilisation des autres méthodes de tokenization compte tenu de la lenteur pour créer un classificateur avec Keras

3. Améliorations

Comme les textes fournis dans le cadre de la compétition *EmoContext* sont des conversations réelles qui utilisent plusieurs *slangs*, acronymes du web et d'autres formes informelles de la langue anglaise, un travail de prétraitement des textes pourrait être la première étape d'amélioration pour ce projet.

Il faudrait alors extraire les slangs du texte et faire une analyse manuellement de la meilleure façon de homogéniser le texte. Par exemple, on pourrait transformer les *slangs* dans les bons mots de la langue, ou si ces acronymes sont trop présents, les considérer comme la langue de base et transformer tous les bons mots dans le *slang* présenté..

Des bibliothèques comme *flashtext* (https://github.com/vi3k6i5/flashtext) pourraient être utiles pour ce travail de normalisation, en utilisation conjointe avec un dictionnaire de *slangs* trouvé sur le web.

L'inefficacité de suppression des *stop words* et l'utilisation des fréquences minimales et maximales pourraient avoir de meilleurs résultats lorsque la normalisation du texte sera en place.

Après avoir un texte normalisé, il serait pertinent d'appliquer un traitement de lemmatisation ou stemming principalement pour les classificateurs SkLearn ou les classificateurs Keras, sauf ceux utilisant Word2Vec.

Du côté de SkLearn, sauf la normalisation des textes, il serait bien d'essayer d'autres modèles, ou même les configurations des classificateurs déjà utilisés pour voir s'il y aurait du changement.

Un classificateur en particulier a soulevé un avertissement lors de l'exécution du code, faudrait l'identifier et puis corriger le problème.

Pour Keras, il y a plusieurs d'autres améliorations qui pourraient être réalisées comme l'ajout de la couche *Convolutional Neural Networks (CNN)* qui a mentionné dans plusieurs articles sur le Web pour ce type de problématique.

Le tokenizer disponible dans la bibliothèque Keras a été utilisé que pour le mode de tokenization aussi. D'autres paramètres sont disponibles pour améliorer la façon dont le code est tokenize, ceux-ci pourraient être testés pour bonifier le projet.